論文93-30B-11-9

목표들간 상호간섭의 분석을 통한 탐색제어 지식의 학습

(Learning Search Control Knowledge From the Analysis of Goal Interactions)

柳 光 烈

(Kwang Ryel Ryu)

要 約

본 논문은 문제풀기에 실패한 경우의 분석을 통해 목표들의 순서정하기 룰을 유도해 낼 수 있도록 하는 방법론을 제시한다. 본방법에서는 실패로 가는 모든 행동들이 분석된다. 취할 수 있는 행동의 종류가 문제의 현재상태로 부터 제약을 받을 경우, 그 제약은 오퍼레이터 바인딩의 제한이라는 형태로 나타나게 된다. 이 관찰을 바탕으로 본 방법은 정확성이 보장되는 일반적 제어룰을 유도해 낸다. 본 방법이 필요로 하는 자료의 양은 탐색트리의 말단 노드들로 부터 나오는 고울 스택들로 한정되어 있으므로, 학습에 소요되는 총 경비가 매우 저렴하다. 실험 결과, 본 방법에 따라 구현된 PAL 시스템이 PRODIGY나 STATIC과 같은 시스템들을 능가하는 성능을 보였다.

Abstract

This paper presents a methodology which enables the derivation of goal ordering rules from the analysis of problem failures. We examine all the possible ways of taking actions that lead to failures. If there are restrictions imposed by a problem state on possible actions to be taken, the restrictions manifest themselves in the form of a restricted set of possible operator bindings. Our method makes use of this observation to derive general control rules which are guaranteed to be correct. The overhead involved in learning is very low because this methodology needs only small amount of data to learn from, namely, the goal stacks from the leaf nodes of a failure search tree. Empirical tests show that the rules derived by our system PAL outperform those derived by other systems such as PRODIGY and STATIC.

I. Introduction

When a problem solver is given a problem with a conjunctive goal condition, perhaps the most difficult task to perform is the ordering of the individual component goals to avoid harmful goal interactions. There have been efforts to cope with this problem either by learning[3] or through reasoning.[1,2] The learning approach taken by systems such as Minton's PRODIGY[3] acquires knowledge for

*正會員, 釜山大學校 컴퓨터工學科
(Dept. of Computer Eng., Pusan Nat'l Univ.)

接受日字 : 1993年 4月 23日

goal ordering by using an EBL (Explanation-Based Learning) technique.[4] The goal ordering rules learned through PRODIGY's search trace analysis, however, are often overly specific, unnecessarily verbose, and sometimes even conflicting. The reasoning approaches taken by Cheng[1] and Etzioni[2] derive goal ordering rules by analyzing the domain operators. However, since their methods are not based on analysis of problem-solving traces, certain constraints specifically imposed by the initial situation of a problem cannot be detected. Etzioni's STATIC system sometimes derives over-general rules which can mislead the planner to generate inefficient plan by expanding extraneous nodes. Another important drawback of the above systems is that they require *a priori* knowledge specific to the problem domain. The PRODIGY system uses such knowledge in its *compression* phase which is critical for enhancing the utility of the learned rules. The STATIC system and Cheng's system rely on domain-specific knowledge when they reason about the effects of the operators. Whenever a new problem domain is given, it is not trivial to determine *a priori* what type of knowledge is needed and in which particular form.

In this paper, we present a new learning method which can derive goal ordering rules from the analysis of problem failures without relying on any *a priori* knowledge. The rules derived by our method are guaranteed to be correct. We also report test results of our implemented planning and learning system called PAL on PRODIGY's test domains of Blocksworld, Stripsworld, Schedworld (machine scheduling), and ABworld (augmented blocks world). The rules derived by PAL outperformed other systems in these domains, while the overhead of learning in PAL is shown to be the lowest.

In the following, we first review the goal interaction problem and discuss the limitations of previous approaches. Then, we give an overview of the PAL's planner and present PAL's learner to describe how it learns from failure analysis. Next comes the experimental results comparing PAL with PRODIGY and STATIC. Other related works are then briefly discussed before the concluding emarks.

Ⅱ. Goal Interaction Problem

The goal interaction refers to the phenomena where the actions for achieving the individual component goals interfere with one another resulting in an inefficient plan. Harmful interactions are avoided if the goals are attacked in a right order. The challenging part of learning goal ordering rules is the determination of the condition under which a certain ordering should be followed. Previous approaches exhibit limitations in deriving this condition as illustrated by an example in the following.

In Stripsworld, the condition for goal ordering is sometimes represented by a complicated relational concept. When the goals are (*status dx closed*) and (*inroom ROB rx*), for example, *ROB*(robot) must get into *rx before* closing *dx* if *dx* is the *only* doorway to *rx*. On the other hand, if *dx* is a door not connected to *rx*, *ROB* should get into *rx after* closing *dx*. The control rule learned from the first case should contain in its antecedent the following condition in addition to the goals:

$$\forall d(connects\ d\ rx\ ra) \rightarrow d{=}dx$$

Similarly, the rule learned from the second case should have the following:

$$\forall r(connects\ dx\ r\ rb) \rightarrow r \neq rx$$

STATIC fails to derive these conditions and constructs over-general rules, one of which prefers (*inroom ROB rx*) to (*status dx closed*) regardless of the location of *dx*. PRODIGY

derives over-specific conditions which overly restrict the applicability of the rules. Cheng's system derives the right conditions with the help of domain-specific knowledge. PAL is able to derive those conditions through analysis and generalization of the goal stacks retrieved from the failure search tree.

Ⅲ. PAL's Planner

PAL's planner employs *means-ends analysis* [5] as its basic search strategy. Given a problem with a conjunctive goal condition, PAL arranges the component goals in the goal stack in the order presented or as recommended by learned rules so that the goals can be processed in that order.

When a goal is true in a state, the goal is removed from the goal stack. Otherwise, for each instantiation of each of the goal's relevant operators, a new copy of the goal stack is created and the instantiated operator together with its preconditions is posted on the goal stack. These preconditions become subgoals and are treated in the same way as the top level goals. An operator in a goal stack is applied when all its preconditions are satisfied. If an operator is applied, it is deleted from the goal stack together with the goal or subgoal that it achieves.

This process recurs until an empty goal stack is found (success) or a dead-end is reached (failure).

1. Operator Representation and Control Heuristics

In PAL's planning model, the literals used in plan representation are separated into two categories, *static* and *dynamic*. The static literals in a state description can never be changed throughout legal state transitions made by operator applications. Dynamic literals are those that can be deleted and/or added by operators.

Figure 1 illustrates the six-slot operator representation used in PAL. Most of the

```
pushthrudr (?b ?d ?rx)
(sp) (pushable ?b) (connects ?d ?ry ?rx)
(bp)
(dp) (status ?d open) (inroom ?b ?ry) (nextto ?b ?d)
     (nextto ROB ?b) (inroom ROB ?ry)
(dl) (nextto ROB ?$) (nextto ?b ?$)
     (nextto ?$ ?b)(inroom ROB ?$) (inroom ?b ?$)
(ma) (inroom ?b ?rx)
(sa) (inroom ROB ?rx) (nextto ROB ?b)

unstack (?x ?y)
(sp)
(bp) (on ?x ?y)
(dp) (clear ?x) (arm-empty)
(dl) (on ?x ?y) (clear ?x)
     (arm-empty)
(ma) (holding ?x) (clear ?y)
(sa)
```

Figure 1. Example operators used in PAL.

default search control heuristics are incorporated into this representation scheme. The first three slots are for the preconditions, the *dl* slot is for the delete list, and the *ma* and *sa* slots are for the add list each containing major effect and side effect of the operator, respectively. The static literals of the preconditions are separately kept in the *sp* slot. When a static precondition of an operator is unsatisfied, the operator is immediately abandoned because it can never be made applicable. The preconditions in *bp* slot, although dynamic, play the same role as static preconditions to heuristically restrict the selection of operators. Only the dynamic preconditions in *dp* slot can become subgoals when they are unsatisfied. An operator is considered relevant to a goal only when it has a literal in its ma slot which unifies with the goal. This easily implements the kinds of heuristics that prevent object-moving operators from being used to move the robot, as can be seen in the *pushthrudr* operator.

2. Strategy for Handling Goal Interactions

PAL encounters harmful goal interactions in two different forms, i.e., protection violation and prerequisite violation. A protection violation is detected when an application of an operator deletes a previously achieved goal. PAL considers a protection violation as a failure and so backtracks. A prerequisite violation is detected when a precondition of an operator is found deleted by another operator applied for a previous goal. A prerequisite violation, however, is considered a failure only when the violated precondition is impossible to be re-achieved. Such irrecoverable prerequisite violations are

frequently observed in the Schedworld domain. If violations are unavoidable with a given ordering of goals, PAL re-orders the goals in such a way that the observed violations are avoided in the new order.

Ⅳ. Learning from Goal Interactions

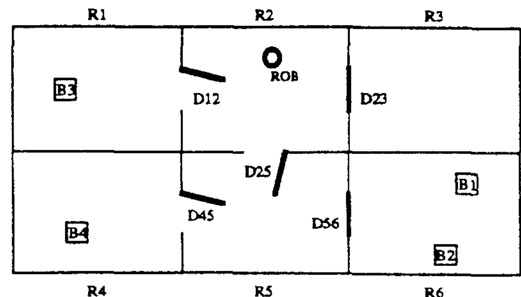
PAL learns goal ordering rules if goals attacked in a certain order lead to failures which involve violations. The most challenging part of learning is the derivation of the general condition under which a certain ordering fails. The central idea of our method is to take into account the mechanism of binding during failure analysis. We can understand how certain harmful interactions are unavoidable in a given situation by examining how all possible ways of taking actions lead to dead ends. If there are restrictions imposed by the problem state on taking possible actions, then the restrictions manifest themselves in the form of a restricted set of possible bindings of operators. Our method makes use of these observations to derive general control rules without relying on any *a priori* domain-specific knowledge. Due to space limitation, we describe our method for the case of learning from two-goal problems, although the method can be applied to *n*-goal problems.

1. Mechanism of Operator Instantiation

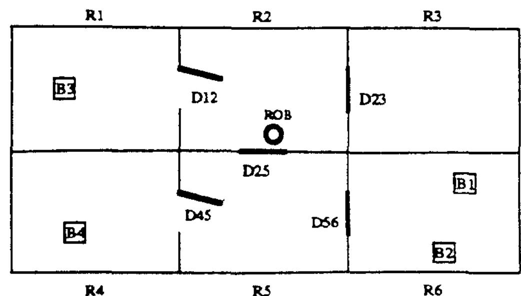
When an operator relevant to a goal is instantiated, an initial binding is first generated by initial matching of the goal with an add literal. This binding is later completed by matching preconditions with the current state description. A dynamic precondition under a certain binding becomes a subgoal if it is not satisfied by the state under that binding. Given a dynamic precondition p of an operator *op*, a set *D* of preconditions other than *p* is called a determinant for p with respect to an add literal *a* of *op* if all the variables appearing in *p* except those in *a* appear in *D*. For example, the set of two literals in the *sp* slot of the *pushthrudr* operator in Figure 1 is a determinant for any precondition in the *dp* slot with respect to the add literal in *ma*. When *pushthrudr* is considered for a goal, the initial matching and the matching of *sp* slot against the state determines all possible bindings. PAL assumes that for any operator *op*, its *sp* together with the *bp* slot constitute a determinant for any dynamic precondition in the *dp* slot of op with respect to any add literal in its *ma* slot.

2. Goal Stack Analysis

The reason why all attempts to achieve a subgoal lead to failures is well recorded in the goal stacks of the failure search tree. Consider a simple problem with the initial state of Figure 2 (a) and the goals g_1 = (*status D25 closed*) and g_2 = (*inroom B1 R2*).



(a)



(b)

Figure 2. Example states in the robot navigation domain.

Figure 2 (b) is the state after g_1 is achieved. From this state every attempt to achieve g_2 either leads to a goal stack cycle (failure by goal repetition) or violation of g_1 because *D25* is the only pathway for *B1* to be moved into *R2*. PAL extracts and generalizes this configurational constraint and the relative location of the box *B1* by investigating the failure goal stacks.

Figure 3 shows the goal stacks retrieved from the failure search tree obtained during the attempt to achieve g_2 starting with the state of Figure 2 (b). Each goal stack is divided into sections for easy reference. The section containing the top level goal is called the *base section*. The remaining part is uniformly divided into sections each of which consists of an operator and its unsatisfied preconditions. They are called the *first section*, *second section*, and so on as indicated in the figure. The subgoal at the top of each section (note the goal stacks are shown upside down) is said to be *active*. The fourth goal stack contains in its first section the active subgoal (*status D23 open*). It was not achieved in this goal stack because opening *D23* was tried from inside *R3* rather than *R2*. In some other search branch, however, it was achieved by selecting the right binding. An active subgoal of this type is said to be *achievable*. The active subgoal (*inroom B1 R3*) of the first goal stack was never achieved in any search branch. This type of active subgoal is called *unachievable*. The unachievable active subgoals constitute the real reason that the top level goal (*inroom B1 R2*) was never achieved. A failure goal stack whose active subgoals are all unachievable is considered to represent an instance of an *unavoidable* failure. Only from the goal stacks of all the unavoidable failures, can we construct an explanation of how all the alternative attempts to achieve a goal lead to failures. In our example, only the first three goal stacks are used for learning.

Let K be a set of all the unavoidable failure goal stacks from the failure search tree obtained during the attempt to achieve the second goal after achieving the first goal. Let e^i_j denote the j-th section of a goal stack $k_i \in K$. The operators and the preconditions in the goal stacks are all given creation during search. Two sections of different goal stacks are identical if the operators and the ordered preconditions in the two sections have the same identifications.

Lemma 1 Let k_i and k_h be two different goal stacks in K. If the active subgoals of e^i_j and e^h_m are identical for some $j, m \geq 1$, *then* $j = m$ *and* $e^i_n = e^h_n$ *for all* $1 \leq n \leq j$

The above lemma states that the goal stacks generated from the same goal stack must have identical sections from the base up

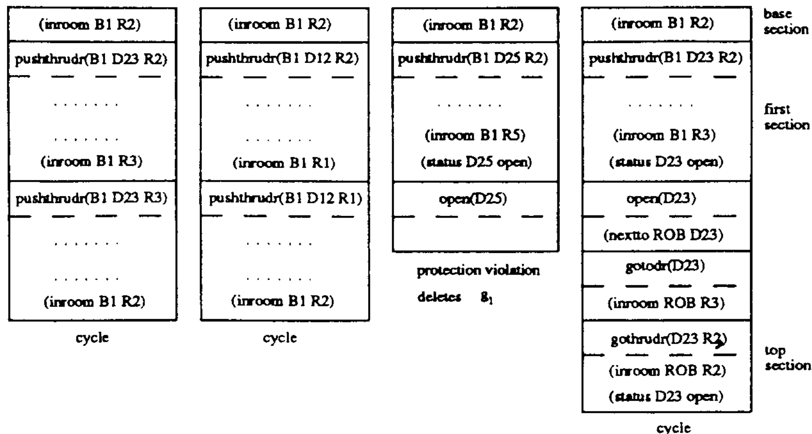
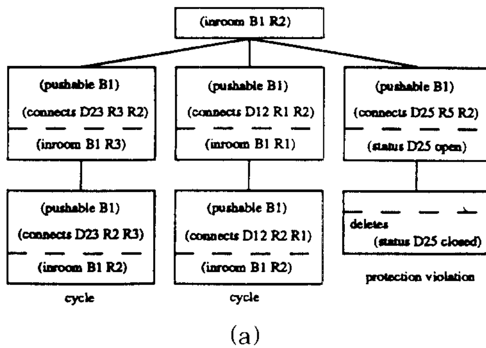


Figure 3. Goal stacks retrieved from a failure search tree.

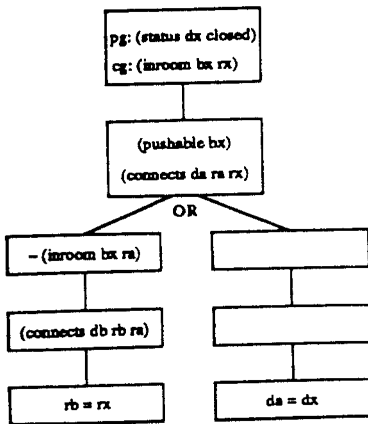
to a certain level.

Lemma 2 Let *p* be an active subgoal of the *j*-th section which is not the top section of a goal stack in *K*. Then, each of all the instantiated relevant operators considered for *p* during search appears in the *(j+1)*-th section of a goal stack in *K*.

None of the operators relevant to *p* is ever successfully applied because otherwise *p* cannot be an unachievable subgoal. Each of those operators must have at least one precondition which is also unachievable and is found in one of the goal stacks in *K*. The goal stacks in *K* show all the unsuccessful attempts to achieve goals from the top level to the lowest level until failures are explicitly detected. Based on the fact that the goal stacks of alternative attempts share identical portions from the base until the sections of different attempts (lemma 1), PAL re-arranges these stacks into a tree structure. We can view each goal stack as a chain of sections with each of the sections connected by edges, starting from the base to the top section. The base section, shared by all the goal stacks, becomes the root of a tree to be constructed. Then, starting from the first section for each level, any identical sections seen at the level are merged into a single node and the sections at the next level are made children of the merged node.



(a)



(b)

Figure 4. The C-tree and F-tree of our example.

Figure 4 (a) shows the tree (called C-tree) derived from our failure goal stacks. Each node in the tree corresponds to a section of a goal stack. However, in each node of the tree, the operators are replaced by their determinants (*sp* and *bp* slots) and the preconditions other than the active ones are simply removed. The determinant in a node can be considered a condition under which the node's subgoal emerges given the goal of its parent node. The leaf node of protection violation does not contain any determinant because the deletion causing the violation is solely determined by the initial binding. This node does not contain a subgoal either.

Instead, it contains the literal which is deleted.

3. Generalization

PAL generalizes the C-tree by substituting constants with variable symbols starting from the root in a top-down manner. The generalization process can be viewed as the reverse of the operator instantiation. First, the goal at the root is variabilized and the constant/variable substitution used is passed down and applied to the children nodes. This is the reverse of the initial binding generated when the relevant operators for the goal are instantiated. The already achieved goal (g_l) is also variabilized and kept separately. Next, the determinants of sibling nodes under the root are generalized to look identical if they

are just different instantiations of a determinant of one operator. Once a determinant of a node is generalized, the subgoal in the same node can be accordingly generalized by applying the constant/variable substitution used in variabilizing the determinant. After all the nodes at the same level are generalized, the nodes are split into parent and child with the determinant as a parent and the subgoal as a child. Negation symbols are attached in front of the generalized subgoals to reflect the fact that they are not satisfied. All the determinants generalized to look identical are merged into a single node moving their children (subgoals) under the merged node. Any identical subgoal nodes are also merged at this time and different children are put under an OR branch to indicate the fact that different subgoals can be created from a single operator depending on bindings. Then, only the substitutions used for variabilizing the subgoals are passed down and applied to the nodes at the next level and the same process is repeated. The children of a subgoal node are always put under an AND branch because *all* the relevant operators for the subgoal should lead to failures. When a leaf node is generalized, the repeated subgoal (cycle failure) and violated goal are replaced by the respective conditions for failure. Such failure conditions are easily derived by examining the repeated subgoal and the delete/add lists of the operator involved in the violation.

As a final step, PAL removes redundant literals from the resulting tree. A literal in a determinant node is redundant if it already appears in one of its ancestor nodes. A literal $\neg p$ in a subgoal node is redundant if p is deleted by every operator which achieves $g1$. The reason is that our tree is intended to represent the condition on the state after $g1$ is achieved. In our example, $\neg$(*status da open*) is redundant under the condition $da = dx$ because the operator achieving the violated goal (*status dx closed*) deletes it. PAL removes redundancy without relying on any domain specific knowledge. The tree after generalization, called an F-tree, is shown in Figure 4 (b). The pg and cg in the root represent the previously achieved goal (g_1) and the current goal (g_2), respectively.

When an F-tree is matched against a new problem, an initial binding is first generated by matching the two goals in the root with the problem goals. This initial binding is then passed down to children nodes for further matching. When a node is matched, the partial bindings passed from its parent are used and extended by matching new variables in the node. When a set of partial bindings generated in a node is passed down through AND branches, all the partial bindings must be successfully completed in every subtree under the branches. If it is through OR branches, each partial binding must be completed in at least one subtree under the branches.

Otherwise, the matching is considered to have failed. The interpretation of the F-tree of Figure 4 (b) is given in the following in the self-explanatory PRODIGY's rule format.

```
(and (current-node n2)
  (candidate-goal n2 (status dx closed))
  (candidate-goal n2 (inroom bx rx))
  (known n2 (pushable bx))
  (for-all (da ra) such-that (known n2
                              (connects da ra rx))
    (or (is-equal da dx)
      (and (known n2 (not (inroom bx ra)))
        (for-all (rb) such-that (known n2
                                 (connects db rb ra))
          (is-equal rb rx))))))
```

If an F-tree matches the state after achieving the first goal of a problem, all the attempts to achieve the next goal will confront the same type of failures summarized in the F-tree.

Theorem 1 Consider a problem with two

ordered goals g_a and g_b. If the state after achieving g_a matches an F-tree, then g_b cannot be achieved from that state provided that the given operators are coherent.

This theorem, however, allows us to use an F-tree only after the first goal of a problem is achieved. For an F-tree to be more useful, it should be possible to use it even before the first goal is achieved. For this, an F-tree has to be *isolated* as explained in the following. Given a goal g, a subset A of given operators is called the *GAC* (goal achievement closure) of g, if (1) all the operators that can achieve g are in A, (2) all the operators that can achieve any precondition of an operator in A are also in A, and (3) A is the minimal such set. An F-tree is said to be *isolated* if none of the generalized subgoals in the tree can possibly be achieved by any operator in the *GAC* of the previously achieved goal.

Theorem 2 If an F-tree which matches an initial state of a problem with ordered pair of goals is isolated, it matches the state after achieving the first goal.

The reason is that the operators applied for the first goal are those in the *GAC* of the first goal and thus none of them can achieve the subgoals in the F-tree. Then, by theorem 1, the second goal cannot be achieved from the state reached by achieving the first goal. If an isolated F-tree matches a problem, we want to change the goal ordering. Only the isolated F-trees are qualified as rules. The proofs of the above two theorems are given in .[6]

Ⅴ. Comparison to PRODIGY and STATIC

PAL was trained and tested on the three PRODIGY's domains with the same training and test problem sets used in.[2] Table 1 shows that PAL spent the shortest average learning time (learning overhead in cpu seconds divided by the number of control rules). PAL's learning time includes the overhead of deriving non-isolated F-trees which are abandoned. The numbers in the parentheses show the number of goal ordering rules. PAL learned only the useful rules which contributed to the speedup of the planner.

Table 1. Average learning time and the number of goal ordering rules.

	PRODIGY	STATIC	PAL
Blocksworld	762 ÷ 19 = 40 (4)	9.9 ÷ 18 = 0.6 (9)	3.5 ÷ 7 = 0.5 (7)
Stripsworld	1057 ÷ 30 = 35 (12)	40.5 ÷ 35 = 1.2 (16)	6.1 ÷ 11 = 0.6 (11)
Schedworld	1374 ÷ 37 = 37 (20)	19.4 ÷ 46 = 0.4 (19)	2.4 ÷ 14 = 0.2 (14).

We used PRODIGY's planner as a testbed. We first had the planner equipped with the control rules learned by PRODIGY's EBL, and then replaced the original goal ordering rules successively by STATIC's rules and PAL's rules to compare the performances. In all the three domains, PRODIGY's rules showed relatively poor performance. Table 2 (a) shows that PAL's rules worked most effectively in Blocksworld, generating more optimal (shorter) solutions with less number of expanded nodes. STATIC's poor performance is due to its failure to derive the rule preferring (*on y z*) to (*on x y*), which are goals frequently seen in this domain. STATIC's rules worked well in Schedworld as shown in Table 2 (b). STATIC derives rules once for all

Table 2. Blocksworld and Schedworld test results..

(a) 100 Blocksworld problems

	cpu secs	nodes	sol. length
no ordering	149	1678	674
PRODIGY	148	1689	650
STATIC	142	1596	630
PAL	139	1561	624

(b) 100 Schedworld problems

	cpu secs	nodes
no ordering	1275	6676
PRODIGY	1092	5982
STATIC	577	2532
PAL	886	4402
PAL$^+$	576	2524

without using any training problems, while PAL's learning can be limited by the given training problems. In Schedworld STATIC outperforms PAL. However, when trained further using test problems, PAL does as well as STATIC as shown in the row PAL*. In Schedworld, the solution lengths are not shown because there were many unsolved problems.

Table 3. Stripsworld test results.

	(a) 103 Stripsworld problems			(b) 98 Cheng's problems		
	cpu secs	nodes	sol. length	cpu secs	nodes	sol. length
no ordering	240	4623	1445	871	9346	2381
PRODIGY	240	4598	1436	748	8222	2351
STATIC*	240	4641	1442	825	8939	2361
STATIC	203	3957	1440	712	7933	2341
PAL	239	4641	1442	698	7739	2341

In Stripsworld, STATIC generated a few over-general rules (four out of sixteen) including those mentioned in section II. The effect of over-general rules given a set of test problems can be either beneficial or harmful depending on the distribution of the problems. In Table 3, STATIC* denotes the STATIC rules excluding the over-general rules. The Stripsworld test problems failed to draw conclusive results as we can see from the the first three rows of table (a) where goal ordering made no improvement. STATIC's apparently better performance (fourth row) over PAL is in fact mostly contributed by an over-general rule in solving one problem. When tested on a more interaction-intensive problems used by Cheng [1], PAL's rules showed the best performance as shown in Table 3 (b).

ABworld is an augmented Blocksworld with an extra macro-operator added to introduce more recursions. It has been shown that the PRODIGY's EBL was not successful in ABworld because of its failure to find rules derivable from non-recursive explanations, while STATIC successfully extracted such rules. [2] The control rules learned from recursive explanations tend to have high matching cost. Four goal ordering rules were derived by STATIC in ABworld. Five rules including those four were learned by PAL when given a set of relevant training problems. PAL shows robust performance against recursions because an F-tree derived from deeply recursive goal stacks is not likely to be isolated (see section IV-3), and thus does not contribute any rule.

VI. Conclusion

We have presented a learning algorithm which is both efficient and effective for deriving sufficient conditions for ordering conjunctive goals. We showed that some complicated relational concepts representing conditions for search control can be derived by a relatively simple method using a minimal amount of information collected from a search tree, namely the failure goal stacks. Unlike other systems, PAL does not rely on any *a priori* domain-specific knowledge which is often difficult to provide. One of the drawbacks of PAL's approach is that the generality of the rules it derives are sometimes limited by the specifics of the training problem instances, while STATIC or Cheng's system do not have such dependency. However, PAL can learn rules whose conditions depend on problem state, which is beyond the scope of those two systems. Moreover, PAL does not show the problems of conflict or over-generality observed with PRODIGY and STATIC.

References

[1] Cheng, J. and Irani, K.B., Ordering Problem Subgoals, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 931-936, Detroit, Michigan (1989).

[2] Etzioni, O., STATIC: A Problem-Space Compiler for Prodigy, *Proceedings of*

the Ninth National Conference on Artificial Intelligence, pp. 533-540 (1991).

[3] Minton, S., Qualitative results concerning the utility of explanation-based learning, *Proceedings of the National Conference on Artificial Intelligence*, St. Paul, MN. (1988).

[4] Mitchell, T.M., Keller, R.M., and Kedar-Cabelli, S.T., Explanation-based generalization: a unifying view, *Machine Learning*, 1:47-80 (1986).

[5] Newell, A., Shaw, J.C. and Simon, H.A., *Report on a General Problem-Solving Program*, *Proceedings of International Conference on Information Processing*, pp. 256-264, Paris, France (1960).

[6] Ryu, K.R., Learning Search Control Knowledge for Planning with Conjunctive Goals, *Ph.D dissertation* in EECS Dept. The University of Michigan, 1992.

[7] Sussman, G.J., *A computer model of skill acquisition*, New York: American Elsevier (1975).

著者紹介

柳光烈(正會員)

1956年 10月 27日生. 1979年 서울대학교 전자공학과 학사. 1981年 서울대학교 전자공학과 석사. 1983年 3月 ~ 1984年 8月 충북대학교 공대 전자계산기공학과 전임강사. 1992年 미시간 대학교 컴퓨터 공학 박사. 1992年 3月 ~ 1993年 2月 포드자동차사 과학연구소 연구원. 1993年 3月 ~ 현재 부산대학교 공대 컴퓨터공학과 전임강사. 주관심분야는 인공지능, 기계학습, 지식기반 시스템, 시스템 통합 등임.